Science of Computer Programming 77 (2012) 505–517

Contents lists available at SciVerse ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

Function extraction

Magnus O. Myreen *, Michael J.C. Gordon

Computer Laboratory, University of Cambridge, Cambridge CB3 0FD, United Kingdom

ARTICLE INFO

Article history:
Received 24 January 2009
Received in revised form 24 September 2010
Accepted 4 October 2010
Available online 31 October 2010

Keywords:
Proof automation
Program verification
Theorem proving

ABSTRACT

Low-level imperative programming languages typically have complex operational semantics (e.g. derived from an underlying processor architecture). In this paper, we describe an automatic method for extracting recursive functions from such low-level programs. The functions are derived by formal deduction from the semantics of the programming language. For each function extracted, a proof of correspondence to the original program is automatically constructed. Subsequent program verification can then be done without referring to the details of the low-level programming language semantics at all: it suffices to prove properties of the extracted function. The technique is explained for simple while programs and also for the machine code of a widely used processor. We show how heuristics can enhance the output from the function extractor/decompiler and how the technique aids implementation of a trustworthy compiler. Our tools have been implemented in the HOL4 theorem prover.

1. Introduction

The connection between recursive functions and imperative programs has been well understood ever since the advent of formal methods. For example, McCarthy pointed out [1] that the imperative code below is equivalent to the functions f and g defined to the right of it.

```
y := 1;
while 0 < x do
  y := x * y;
  x := x - 1
end
```

$$\approx$$

$$f(x) = \text{let } y = 1 \text{ in } g(x, y)$$
$$g(x, y) = \text{if } \neg(0 < x) \text{ then } (x, y) \text{ else}$$
$$\text{let } y = x \times y \text{ in}$$
$$\text{let } x = x - 1 \text{ in}$$
$$g(x, y)$$

This correspondence between imperative and functional programs seems to have been largely overlooked for purposes of program verification, although it did underlie some impressive early work by Boyer and Moore [2]. The reason why program verification can benefit from this connection is simple: to prove a property of the imperative program we must refer to its potentially complicated semantics, but proving a property of the extracted functional representation can be significantly simpler, since it only needs standard mathematical techniques. As Boyer and Moore pointed out [2], the method is an alternative to verification conditions.

The contrast and importance of this correspondence becomes far more noticeable for lower-level languages. For example, the ARM machine code below is described by the function f defined to the right of it.

```
E3A00000
E3510000
12800001
15911000
1AFFFFFB
```

$$\approx$$

$$f(r_1, m) = \text{let } r_0 = 0 \text{ in } g(r_0, r_1, m)$$
$$g(r_0, r_1, m) = \text{if } r_1 = 0 \text{ then } (r_0, r_1, m) \text{ else}$$
$$\text{let } r_0 = r_0 + 1 \text{ in}$$
$$\text{let } r_1 = m(r_1) \text{ in}$$
$$g(r_0, r_1, m)$$

* Corresponding author.
E-mail address: magnus.myreen@cl.cam.ac.uk (M.O. Myreen).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.10.001

Reasoning about the execution of this machine code using a formal model of an ARM processor is tricky. Our approach uses a semantics based on an ARM model to generate the function f completely automatically. Proving properties of the execution of the code can then be replaced by the simpler task of proving properties of the corresponding function f, a task that can be done without any knowledge of the machine-code semantics (i.e. the ARM processor ISA).

In this paper, we show how functions, such as those shown above, can be extracted automatically by mechanised deduction using the programming language semantics. For each translation, our algorithm proves a theorem which states that the function accurately describes all possible executions of the imperative program. The work presented here combines and extends material from several earlier conference papers [3–6] into a single coherent presentation (in some of these papers we refer to *function extraction* as *decompilation into logic*). We explain how such translations can be performed on a simple while language (Section 2, [4]) before describing how functions can be extracted from machine code (Section 3, [3]). We also present some new results on a heuristics-enhanced approach to function extraction where the function extractor performs part of the verification proof (Section 4). Our heuristics attempt to extract functions that operate over lists rather than a flat array-like memory; for example, the following function *reverse* is extracted from the pointer manipulating C code below.

```
p = NULL;
while((*i) != NULL) {
  x = (*i)->tl;
  (*i)->tl = p;
  p = (*i);
  (*i) = x;
}
```

$\approx$

$$reverse(xs) = rev(xs, [])$$
$$rev([], ys) = ys$$
$$rev(x :: xs, ys) = rev(xs, x :: ys)$$

This paper then continues with comments on some applications we have explored in the context of trustworthy compilation (Section 5, [5,6]), followed by a discussion of related work (Section 6). The translations described here have been implemented [7] in the HOL4 theorem prover [8].

2. Function extraction for a while language

We start by explaining the main idea behind function extraction informally in terms of a simple while language before presenting, in Section 3, how the idea can be used for formal proofs concerning realistic machine languages.

2.1. Hoare logic

In this section we will assume that $\{p\}\ c\ \{q\}$ is a standard Hoare triple [9] for a simple while language,

$c ::= v := exp \mid c\,;c \mid \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end} \mid \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c$

for variables v, arithmetical expressions *exp*, and Boolean expressions b. We assume that $\{p\}\ c\ \{q\}$ satisfies the standard proof rules of Hoare logic. In fact, we define Hoare triples in terms of an operational semantics of the programming language (which, in Section 3, is a processor model for machine code) and then derive the axioms and rules of Hoare logic from this. The derived Hoare logic is then the basis for our method of function extraction. The details of how Hoare logic is derived from the operational semantics are fairly standard and are not needed for what follows, so are not given here.

2.2. Intermediate format

Our algorithm for function extraction generates Hoare triples using Hoare logic in a bottom-up manner based on the structure of programs. For instance, when McCarthy's example

```
y := 1; while 0 < x do y := x * y; x := x - 1 end
```

is traversed, a Hoare triple theorem will first be proved for each assignment, then the innermost sequential composition will be processed, followed by the while construct, and finally the outermost sequential composition will be processed. At each stage, a Hoare triple theorem of the form shown below is returned. In such generated triples, f is the extracted function, which expects as input the value (x, y). The precondition ensures that these logical variables, x and y, are equal to the initial value of program variables x and y.

$\{(\mathrm{x}, \mathrm{y}) = (x, y)\}\ code\ \{(\mathrm{x}, \mathrm{y}) = f(x, y)\}$

2.3. Assignments

Each assignment is easily turned into a Hoare triple of the desired form; for example, `x := x - 1` can be represented as

$\{(\mathrm{x}, \mathrm{y}) = (x, y)\}$ `x := x - 1` $\{(\mathrm{x}, \mathrm{y}) = (x - 1, y)\}$

It will be useful later to express such postconditions as functions consisting of a let expression applied to the initial values; for example,

$(x - 1, y) = (\lambda(x, y).\ \mathsf{let}\ x = x - 1\ \mathsf{in}\ (x, y))\ (x, y)$

2.4. Sequential composition

The function describing the effect of executing c_1 then c_2 is produced by composing the functions describing each of the two code segments:

$$(\forall x.\ \{r = x\}\ c_1\ \{r = f(x)\}) \wedge$$
$$(\forall x.\ \{r = x\}\ c_2\ \{r = g(x)\}) \Rightarrow$$
$$(\forall x.\ \{r = x\}\ c_1 ; c_2\ \{r = g(f(x))\})$$

2.5. Conditional statements

Conditional statements can be composed together in a similar manner. The following theorem holds if all variables in b occur in variable list r.

$$(\forall x.\ \{r = x\}\ c_1\ \{r = f(x)\}) \wedge$$
$$(\forall x.\ \{r = x\}\ c_2\ \{r = g(x)\}) \Rightarrow$$
$$(\forall x.\ \{r = x\}\ \texttt{if}\ \ b\ \ \texttt{then}\ \ c_1\ \ \texttt{else}\ \ c_2\ \{r = \mathsf{if}\ b[x/r]\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x)\})$$

2.6. Loops

Each loop can be described by a tail-recursive function, i.e. an instance of the following template function:

$$\mathsf{whilefun}(h, g)(x) = \mathsf{if}\ h(x)\ \mathsf{then}\ \mathsf{whilefun}(h, g)(g(x))\ \mathsf{else}\ x$$

Moore and Manolios showed that such a function can be defined in logic without a termination proof [10].

The loop rule uses whilefun to represent the semantics of the `while` programs. Again, all variables in b must occur in variable list r.

$$(\forall x.\ \{r = x\}\ c\ \{r = g(x)\}) \Rightarrow$$
$$(\forall x.\ \{r = x\}\ \texttt{while}\ \ b\ \texttt{do}\ c\ \texttt{end}\ \{r = \mathsf{whilefun}((\lambda x.\ b[x/r]), g)(x)\})$$

This loop rule follows from the standard rule for partial correctness of `while`:

$$\{I \wedge b\}\ c\ \{I\}\ \Rightarrow\ \{I\}\ \texttt{while}\ \ b\ \texttt{do}\ c\ \texttt{end}\ \{I \wedge \neg b\}$$

2.7. Example

A straightforward application of each of the steps described above produces the following function when applied to the loop in McCarthy's example.

$$\mathsf{whilefun}((\lambda(x, y).\ 0 < x),$$
$$\quad (\lambda(x, y).\ \mathsf{let}\ x = x - 1\ \mathsf{in}\ (x, y)) \circ (\lambda(x, y).\ \mathsf{let}\ y = y \times x\ \mathsf{in}\ (x, y)))$$

Although such automatically generated functions are hard to read in their raw form, they can easily be postprocessed into a readable format using a few simple steps that are easy to automate. For example, if we call the above expression g, we can unroll the definition of whilefun to get a readable equation:

$$\begin{aligned}
& g(x, y) \\
&= \mathsf{whilefun}((\lambda(x, y).\ 0 < x), \ldots)(x, y) \\
&= \mathsf{if}\ 0 < x\ \mathsf{then}\ \mathsf{whilefun}((\lambda(x, y).\ 0 < x), \ldots)(x, y)\ \mathsf{else}\ (x, y) \\
&= \mathsf{if}\ \neg(0 < x)\ \mathsf{then}\ (x, y)\ \mathsf{else}\ \mathsf{whilefun}((\lambda(x, y).\ 0 < x), \ldots)((\ldots)(x, y)) \\
&= \mathsf{if}\ \neg(0 < x)\ \mathsf{then}\ (x, y)\ \mathsf{else}\ g((\ldots)(x, y)) \\
&= \mathsf{if}\ \neg(0 < x)\ \mathsf{then}\ (x, y)\ \mathsf{else}\ g(\mathsf{let}\ y = y \times x\ \mathsf{in}\ \mathsf{let}\ x = x - 1\ \mathsf{in}\ (x, y)) \\
&= \mathsf{if}\ \neg(0 < x)\ \mathsf{then}\ (x, y)\ \mathsf{else}\ \mathsf{let}\ y = y \times x\ \mathsf{in}\ \mathsf{let}\ x = x - 1\ \mathsf{in}\ g(x, y)
\end{aligned}$$

We know that this definition of g is an accurate description of the effect of the loop since our algorithm proves the following Hoare triple theorem.

$$\{(\mathtt{x}, \mathtt{y}) = (x, y)\}\ \texttt{while 0 < x do y := y*x; x := x-1 end}\ \{(\mathtt{x}, \mathtt{y}) = g(x, y)\}$$

We call such theorems *certificate theorems* since they certify that function g actually corresponds to the execution of the original program.

2.8. Termination

The loop rule described above assumes that we are dealing with a Hoare triple for partial correctness, i.e. the postcondition holds whenever the program terminates—it does not guarantee that the program terminates.

To prove certificate theorems for total-correctness Hoare triples, we must introduce a side condition into the conclusion of the loop rule: the while rule holds only if the loop guard will eventually terminate, i.e. if there exists a number of iterations after which update g makes guard h false.

$$\exists n.\ \neg h(g^n(x))$$

The introduction of such side conditions has a knock-on effect on the entire algorithm, since each step of the algorithm must now return certificate theorems of the form shown below, where s is a side condition.

$$\forall x.\ s(x)\ \Rightarrow\ \{r = x\}\ c\ \{r = f(x)\}$$

The proof rules for composition and if statements are modified to accommodate side conditions:

$$\begin{array}{l}(\forall x.\ s_1(x) \Rightarrow \{r = x\}\ c_1\ \{r = f(x)\}) \wedge \\ (\forall x.\ s_2(x) \Rightarrow \{r = x\}\ c_2\ \{r = g(x)\}) \Rightarrow \\ (\forall x.\ s_1(x) \wedge s_2(f(x)) \Rightarrow \{r = x\}\ c_1\,;c_2\ \{r = g(f(x))\})\end{array}$$

$$\begin{array}{l}(\forall x.\ s_1(x) \Rightarrow \{r = x\}\ c_1\ \{r = f(x)\}) \wedge \\ (\forall x.\ s_2(x) \Rightarrow \{r = x\}\ c_2\ \{r = g(x)\}) \Rightarrow \\ (\forall x.\ (\mathsf{if}\ b[x/r]\ \mathsf{then}\ s_1(x)\ \mathsf{else}\ s_2(x)) \Rightarrow \\ \quad \{r = x\}\ \texttt{if}\ b\ \texttt{then}\ c_1\ \texttt{else}\ c_2\ \{r = \mathsf{if}\ b[x/r]\ \mathsf{then}\ f(x)\ \mathsf{else}\ g(x)\})\end{array}$$

The side condition introduced by the loop rule is defined to assert that the loop terminates within some number of iterations n, and that the side condition s for the loop body is true, for each iteration of the loop leading up to its termination.

$$\begin{array}{l}\mathsf{whilepre}(g, h, s)(x) = \exists n.\ \neg h(g^n(x)) \wedge \\ \qquad\qquad\qquad\qquad \forall k.\ (\forall m.\ m \leq k \Rightarrow h(g^m(x))) \Rightarrow s(g^k(x))\end{array}$$

This definition of whilepre may seem hard to work with. In practice, it is sufficient to use the following equation (which follows from the definition above by a simple proof based on a case split on whether $h(x)$ is true: if $h(x)$ is true then the loop will require at least one more iteration, i.e. $n \geq 1$).

$$\mathsf{whilepre}(g, h, s)(x) = \mathsf{if}\ h(x)\ \mathsf{then}\ \mathsf{whilepre}(g, h, s)(g(x)) \wedge s(x)\ \mathsf{else}\ \mathsf{true}$$

The loop rule for generating total-correctness Hoare triples allows the loop body to depend on a side condition s, and produces a certificate theorem with side condition $\mathsf{whilepre}(g, h, s)(x)$:

$$\begin{array}{l}(\forall x.\ s(x) \Rightarrow \{r = x\}\ c\ \{r = g(x)\}) \wedge (h = \lambda x.\ b[x/r]) \Rightarrow \\ (\forall x.\ \mathsf{whilepre}(g, h, s)(x) \Rightarrow \{r = x\}\ \texttt{while}\ b\ \texttt{do}\ c\ \texttt{end}\ \{r = \mathsf{whilefun}(h, g)(x)\})\end{array}$$

The definition of the function f_{pre} below is the automatically generated accumulated side condition for McCarthy's example, when a total-correctness certificate is derived. The derived equation for whilepre is used to unroll the side condition so that it closely resembles the extracted function itself.

$$\begin{array}{rl}f_{pre}(x) &= \mathsf{let}\ y = 1\ \mathsf{in}\ g_{pre}(x, y) \\ g_{pre}(x, y) &= \mathsf{if}\ \neg(0 < x)\ \mathsf{then}\ \mathsf{true}\ \mathsf{else} \\ &\quad\ \mathsf{let}\ y = x \times y\ \mathsf{in} \\ &\quad\ \mathsf{let}\ x = x - 1\ \mathsf{in} \\ &\quad\ g_{pre}(x, y)\end{array}$$

2.9. Preconditions

Notice that the side conditions mimics very closely the structure of the extracted function, and in fact much of the control structure of f is duplicated in side condition s in certificate theorems of the form

$$\forall x.\ s(x)\ \Rightarrow\ \{r = x\}\ c\ \{r = f(x)\}$$

This observation leads to an important optimisation: we can avoid this duplication by storing the extracted function inside the side condition as an equation. For example, the above is equivalent to the following, if y is a fresh variable name.

$$\forall y\, x.\ s(x) \wedge y = f(x)\ \Rightarrow\ \{r = x\}\ c\ \{r = y\}$$

Composition of such a theorem with a similar theorem for c_2, s_2 and f_2 gives

$$\forall z\, y\, x.\ (s(x) \wedge y = f(x) \wedge s_2(y) \wedge z = f_2(y))\ \Rightarrow\ \{r = x\}\ c\,;c_2\ \{r = z\}$$

Similarly, if statements of such theorems produce if statements in the side condition:

$$\begin{array}{l}\forall y\, x.\ (\mathsf{if}\ b[x/r]\ \mathsf{then}\ s(x) \wedge y = f(x)\ \mathsf{else}\ s_2(x) \wedge y = f_2(x)) \Rightarrow \\ \quad \{r = x\}\ \texttt{if}\ b\ \texttt{then}\ c\ \texttt{else}\ c_2\ \{r = y\}\end{array}$$

This optimisation helps keep expression sizes small in intermediate stages between applications of the loop rule. Before each application of the loop rule, we separate the side conditions from the encoded function.

3. Decompiling machine code

So far, we have considered a toy while language. In this section, we turn our attention to realistic low-level languages: machine code. By reasoning down to the level of real machine code, we introduces several mathematically unclean features that are not present in simplified high-level languages.

A. *On real machines all types are bounded*: memory, stacks, and even integers are bounded; as a consequence, programs cannot assume an arbitrarily large stack. Furthermore, some familiar arithmetic properties do not hold for machine integers; for example, it is not the case that $\forall v\ w.\ 0 \leq w \Rightarrow v \leq v + w$, if v and w are machine integers.

B. *Machine code operates over a heterogeneous state*, which consists of various machine-specific registers, status bits/flags, special-purpose registers, memory segments, and operation modes, instead of a straightforward mapping from variable names to values.

C. *Machine code is generally less structured than high-level languages*: Individual instructions rarely execute a single assignment; most instructions update registers as well as status bits and possibly also make memory accesses. Code and data live in the same memory, meaning that programs can accidentally (or intentionally) rewrite themselves.[1] Control flow is determined by updates to a register (the program counter), which holds the address of the next instruction to be executed.

In this section, we will explain how our technique for function extraction can be adapted to cope with the challenges posed by these differences. The extracted functions will now operate over various finite bitvector types instead of basic integers. Certificate theorems become more challenging to state formally and prove, due to B and C. The loop rule also needs to be generalised to deal with the new forms of control flow.

3.1. Hoare logic for machine code

We start by presenting a total-correctness Hoare triple, which we use for the certificate theorems produced by the function extractor. Our machine-code Hoare triples have been instantiated to realistic models of ARM, x86, and PowerPC architectures developed by Fox [11], Sarkar et al. [12], and Leroy [13], respectively. We give the top-level definition of our machine-code Hoare triple first and then explain in detail how states and code are represented, and the meaning of the associated operators $*$, set, and code.

The machine-code Hoare triple $\{p\}\ c\ \{q\}$ is true whenever any state s, which satisfies precondition p and has code c in memory, can be transformed, by some k-steps of execution of the operational semantics next, into a state which satisfies postcondition q and has code c in memory.

$$\begin{aligned}\{p\}\ c\ \{q\} = \forall s\ r.\ (p * \mathsf{code}\ c * r)\ (\mathsf{set}(s)) \Rightarrow \\ \exists k.\ (q * \mathsf{code}\ c * r)\ (\mathsf{set}(\mathsf{next}^k(s)))\end{aligned} \tag{1}$$

This definition incorporates the ideas of 'local reasoning' from separation logic by supporting the frame rule. The frame rule allows any assertion r to be appended to the precondition and the postcondition using '$*$', which operates on sets representing states (explained below).

$$\{p\}\ c\ \{q\} \Rightarrow \forall r.\ \{p * r\}\ c\ \{q * r\}$$

The $*$ operator is called a separating conjunction. Informally, $p * q$ is true for a state, if that state can be split into two disjoint substates so that p is true for one substate and q for the other. We define the $*$ operator over sets

$$(p * q)\ s = \exists u\ v.\ p\ u \wedge q\ v \wedge (u \cup v = s) \wedge (u \cap v = \{\})$$

and use a function set to translate states into an appropriate set representation. Each state is represented as a set of state components; for example, the following set could be the representation of a machine state where register 0 has value 5, register 1 value 60, and similarly memory location 4000 contains value 3, etc.

$$\{\mathsf{Reg}\ 0\ 5, \mathsf{Reg}\ 1\ 60, \ \ldots, \ \mathsf{Mem}\ 4000\ 3, \ \ldots\}$$

For each architecture, a separate translation function called set is defined. Such translation functions take an input *state* and produce a set of state elements, i.e. each state translates into a set where each register, memory location, etc. is represented by a separate element. For example, for ARM we have the following of the translation function set.

$$\begin{aligned}&\mathsf{set}\ \mathit{state} = \\ &\quad \{\mathsf{Reg}\ r\ (\mathsf{arm\_read\_reg}\ r\ \mathit{state})) \mid r \in \mathsf{arm\_regs}\} \cup \\ &\quad \{\mathsf{Mem}\ a\ (\mathsf{arm\_read\_mem}\ a\ \mathit{state})) \mid a \in \mathsf{arm\_mem\_addresses}\} \cup \ \ldots\end{aligned}$$

Here arm_read_reg and arm_read_mem read the value of a register and a memory location, respectively, with respect to the current operation mode.

[1] Most modern operating systems will raise an exception in certain cases of unintended self-modification.

The separating conjunction is called 'separating' since it asserts that resources are disjoint, as we shall see below. Let R r x assert that register r has value x and M a y assert that memory location a contains y.

$$(\mathsf{M}\ a\ w)\ s = (s = \{\mathsf{Mem}\ a\ w\})$$
$$(\mathsf{R}\ r\ v)\ s = (s = \{\mathsf{Reg}\ a\ v\})$$

We observe from the following that $*$ separates assertions:

$$\forall p\ s.\ (\mathsf{M}\ a\ w_1 * \mathsf{M}\ b\ w_2 * \mathsf{R}\ r\ w_3 * \mathsf{R}\ d\ w_4 * p)\ (\mathsf{set}\ s) \Rightarrow a \neq b\ \wedge\ r \neq d$$

We will abbreviate R 1, R 2 as r1, r2, etc.

A program c is represented as a sets of pairs (a, i), where i is an instruction and a a memory address. In the definition of the machine-code Hoare triple (1), code c asserts, for each pair $(a, i) \in c$, that i is located at address a. Note that with this representation, if c_1 and c_2 are programs, then $c_1 \cup c_2$ is their combination. Here $\cup$ is set union. The composition rule (2), which will be explained in Section 3.3, uses set union. In the definition of the function code below, [31–2] is a function which selects the upper 30 bits of a 32-bit word.

$$(\mathsf{code}\ c)\ s = (s = \{\mathsf{Mem}\ (a[31{-}2])\ i \mid (a, i) \in c\})$$

Note that both the precondition and the postcondition of our Hoare-triple definition are asserted separately from the code assertion code c, and thus implicitly assume that the code is separate from any memory locations mentioned in the precondition or the postcondition. A few examples in the next section will illustrate this.

The assertion m m states that each aligned address a in the domain of m (i.e. a is a 32-bit word which satisfies $a \,\&\, 3 = 0$ and $a \in \mathsf{domain}\ m$) contains the value $m(a)$.

$$(\mathsf{m}\ m)\ s = (s = \{\mathsf{Mem}\ (a[31{-}2])\ (m(a)) \mid a \in \mathsf{domain}\ m \wedge a \,\&\, 3 = 0\})$$

Note that m does not necessarily model the entire memory. For example, it is possible to have two m assertions m m and m h that model disjoint memory areas (i.e. m m $*$ m h, which entails domain $m \cap$ domain $h = \{\}$).

3.2. Hoare triple examples

A few examples of machine-code Hoare triples will show how they look in practice. The effect of the ARM instruction for subtracting 50 from the value of register 2 is described by the following Hoare triple: if register 2 initially has value x and the program counter has value p, which points at instruction '`sub r2,r2,#50`', then register 2 will get value $x{-}50$ and the program counter will be incremented by the length of the instruction, i.e. 4 bytes.

$$\{\mathsf{r2}\ x * \mathsf{pc}\ p\}$$
$$p : \texttt{E1056096 [sub r2,r2,\#50]}$$
$$\{\mathsf{r2}\ (x{-}50) * \mathsf{pc}\ (p{+}4)\}$$

Such theorems implicitly state that nothing else changed due to the frame rule, which was mentioned above. The frame rule allows any assertion to be added to both the precondition and the postcondition; for example, we can add the assertion 'register 7 has value z', which we write formally as r7 z, to both the precondition and the postcondition.

The ARM instruction for swapping the value of a memory word from an address stored in register 6 into register 5 is described by the following Hoare triple, which requires a side condition on the address which accesses memory (since this instruction performs a word sized load/store, the address must be word aligned $x \,\&\, 3 = 0$). The address must also be part of the memory segment described by m; i.e. $x \in$ domain m.

$$\{\mathsf{r5}\ x * \mathsf{r6}\ y * \mathsf{m}\ m * \mathsf{pc}\ p * \langle x \,\&\, 3 = 0 \wedge x \in \mathsf{domain}\ m\rangle\}$$
$$p : \texttt{E1056096 [swp r6,r6,[r5]]}$$
$$\{\mathsf{r5}\ x * \mathsf{r6}\ (m(x)) * \mathsf{m}\ (m[x \mapsto y]) * \mathsf{pc}\ (p{+}4)\}$$

Note that we need not state explicitly that the memory update does not update the code since m is mentioned in the precondition which is guaranteed to be separate from the code by the definition of the machine-code Hoare triple.

3.3. Function extraction and loop rule for machine code

The algorithm for function extraction from machine code (also called decompilation) follows very closely the algorithm for extraction applied to a while language. Certificate theorems from intermediate stages are composed using the following composition rule. The intuition for this rule is as follows: if instructions c_1 are sufficient to transform any state satisfying p into one satisfying q, and c_2 similarly from q to r, then these instructions will together ($c_1 \cup c_2$) be sufficient to transform any state satisfying p into one satisfying r.

$$\{p\}\ c_1\ \{q\}\ \wedge\ \{q\}\ c_2\ \{r\}\ \Rightarrow\ \{p\}\ (c_1 \cup c_2)\ \{r\} \tag{2}$$

The loop rule for machine code is conceptually only different from the loop rule for the while language described earlier in a minor detail: on exit while loops in the while language do not alter the state; in contrast, loops in machine code might modify the state on exit. Thus we introduce a function d to keep track of how the state is modified when the code exits a loop. Each loop in machine code can be described by a tail-recursive function of the form

$$\mathsf{tailrec}(h, g, d)(x) = \mathsf{if}\ h(x)\ \mathsf{then}\ \mathsf{tailrec}(h, g, d)(g(x))\ \mathsf{else}\ d(x)$$

One can define tailrec in terms of whilefun:

$$\mathsf{tailrec}(h, g, d)(x) = d(\mathsf{whilefun}(h, g)(x))$$

The side condition produced by the machine-code loop rule is also slightly different: the side condition s, which must be true for each iteration of the loop, must now be true also in the case when the loop terminates. We call the loop rule's side condition tailrec_pre. It satisfies

$$\mathsf{tailrec\_pre}(g, h, s)(x) = \mathsf{if}\ h(x)\ \mathsf{then}\ \mathsf{tailrec\_pre}(g, h, s)(g(x)) \wedge s(x)\ \mathsf{else}\ s(x)$$

We define tailrec_pre formally as follows:

$$\begin{aligned}\mathsf{tailrec\_pre}(g, h, s)(x) = {} & \exists n.\ \neg h(g^n(x)) \wedge \\ & \forall k.\ (\forall m.\ m < k \Rightarrow h(g^m(x))) \Rightarrow s(g^k(x))\end{aligned}$$

The loop rule for machine code is the following theorem: if for all x such that $s(x)$, each execution of code c from a state $p(x)$ will reach either a state $p(g(x))$ or $q(d(x))$ depending on the truth of $h(x)$, then for any x for which tailrec_pre is true, execution of c from a state satisfying $p(x)$ will reach a state where tailrec describes the result.

$$\begin{aligned}&(\forall x.\ s(x) \Rightarrow \{p(x)\}\ c\ \{\mathsf{if}\ h(x)\ \mathsf{then}\ p(g(x))\ \mathsf{else}\ q(d(x))\}) \Rightarrow \\ &(\forall x.\ \mathsf{tailrec\_pre}(h, g, s)(x) \Rightarrow \{p(x)\}\ c\ \{q(\mathsf{tailrec}(h, g, d)(x))\})\end{aligned}$$

3.4. Example: non-decomposable loop

We illustrate the loop rule for machine code with an example of a non-decomposable loop, i.e. code which consists of two loops neither of which is nested within the other. This loop, in ARM machine code, has two entry points identified by labels L and G. Understanding the exact effect of each instruction is not relevant for understanding this example.

```
 0:  010080E2     L:  add  r0,r0,#1
 4:  010010E3     G:  tst  r0,#1
 8:  FCFFFF1A         bne  L          /* might jump to L */
12:  020050E2         subs r0,r0,#2
16:  FBFFFF1A         bne  G          /* might jump to G */
```

In order to apply the loop rule, we need a Hoare-triple theorem describing the effect of one pass through this code. The first step is to identify all possible paths through the code, namely

$$0 \to 4 \qquad 4 \to 8 \to 0 \qquad 4 \to 8 \to 12 \to 16 \to 4 \qquad 4 \to 8 \to 12 \to 16 \to 20$$

We then compose the Hoare triples for each instruction to get a Hoare-triple theorem for each path. Let code be the set of machine instructions listed above.

$$\{\mathsf{r1}\ x * \mathsf{pc}\ (p{+}0)\}\ \mathsf{code}\ \{\mathsf{r1}\ (x{+}1) * \mathsf{pc}\ (p{+}4)\}$$

$$\{\mathsf{r1}\ x * \mathsf{pc}\ (p{+}4)\}\ \mathsf{code}\ \{\mathsf{r1}\ x * \mathsf{pc}\ (p{+}0)\} \qquad \text{if } x \,\&\, 1 = 0$$

$$\{\mathsf{r1}\ x * \mathsf{pc}\ (p{+}4)\}\ \mathsf{code}\ \{\mathsf{r1}\ (x{-}2) * \mathsf{pc}\ (p{+}4)\} \qquad \text{if } x \,\&\, 1 \neq 0 \text{ and } x{-}2 \neq 0$$

$$\{\mathsf{r1}\ x * \mathsf{pc}\ (p{+}4)\}\ \mathsf{code}\ \{\mathsf{r1}\ (x{-}2) * \mathsf{pc}\ (p{+}20)\} \qquad \text{if } x \,\&\, 1 \neq 0 \text{ and } x{-}2 = 0$$

As a next step, we automatically generate a precondition pre, a postcondition post, and component functions h, g, and d. We use a position argument pos to specify which entry point is used: $pos = 0$ is equivalent to label L in the code.

$$\begin{aligned}\mathsf{pre}(x, pos) &= \mathsf{r1}\ x * \mathsf{pc}\ (\mathsf{if}\ pos = 0\ \mathsf{then}\ p\ \mathsf{else}\ p{+}4) \\ \mathsf{post}(x) &= \mathsf{r1}\ x * \mathsf{pc}\ (p{+}20) \\ \mathsf{h}(x, pos) &= \mathsf{if}\ pos = 0\ \mathsf{then\ true\ else} \\ &\qquad \mathsf{if}\ x \,\&\, 1 = 0\ \mathsf{then\ true\ else} \\ &\qquad\ \mathsf{if}\ x{-}2 \neq 0\ \mathsf{then\ true\ else\ false} \\ \mathsf{g}(x, pos) &= \mathsf{if}\ pos = 0\ \mathsf{then}\ (x{+}1, 1)\ \mathsf{else} \\ &\qquad \mathsf{if}\ x \,\&\, 1 = 0\ \mathsf{then}\ (x, 0)\ \mathsf{else} \\ &\qquad\ \mathsf{if}\ x{-}2 \neq 0\ \mathsf{then}\ (x{-}2, 1)\ \mathsf{else\ arb} \\ \mathsf{d}(x, pos) &= \mathsf{if}\ pos = 0\ \mathsf{then\ arb\ else} \\ &\qquad \mathsf{if}\ x \,\&\, 1 = 0\ \mathsf{then\ arb\ else} \\ &\qquad\ \mathsf{if}\ x{-}2 \neq 0\ \mathsf{then\ arb\ else}\ x{-}2\end{aligned}$$

With these definitions, the Hoare-triple theorems above can be merged into one Hoare-triple theorem. Let $\mathsf{s}(y) = \mathsf{true}$.

$$\forall y.\ \mathsf{s}(y) \Rightarrow \{\mathsf{pre}(y)\}\ \mathsf{code}\ \{\mathsf{if}\ \mathsf{h}(x)\ \mathsf{then}\ \mathsf{pre}(\mathsf{g}(x))\ \mathsf{else}\ \mathsf{post}(\mathsf{d}(x))\}$$

This theorem fits the loop rule, i.e. tailrec describes the effect of the loop:

$$\forall y.\ \mathsf{tailrec\_pre}(\mathsf{h}, \mathsf{g}, \mathsf{s})(y) \Rightarrow \{\mathsf{pre}(y)\}\ \mathsf{code}\ \{\mathsf{post}(\mathsf{tailrec}(\mathsf{h}, \mathsf{g}, \mathsf{d})(y))\}$$

Let the above instances of tailrec and tailrec_pre be called loop and loop_pre. Expansion of definitions for pre and post, and instantiation of y with $(x, 0)$, produces our certificate theorem:

$$\mathsf{loop\_pre}(x, 0) \ \Rightarrow\ \{\mathsf{r1}\ x * \mathsf{pc}\ p\}\ \mathsf{code}\ \{\mathsf{r1}\ (\mathsf{loop}(x, 0)) * \mathsf{pc}\ (p{+}20)\}$$

Unrolling tailrec allows us to form easy to understand equations describing the extracted function loop:

$$\begin{aligned}\mathsf{loop}(x, pos) = \mathsf{if}\ pos = 0\ \mathsf{then}\ \mathsf{loop}(x{+}1, 1)\ \mathsf{else}\\ \mathsf{if}\ x \,\&\, 1 = 0\ \mathsf{then}\ \mathsf{loop}(x, 0)\ \mathsf{else}\\ \mathsf{if}\ x{-}2 \neq 0\ \mathsf{then}\ \mathsf{loop}(x{-}2, 1)\ \mathsf{else}\ x{-}2\end{aligned}$$

Similarly, the termination condition loop_pre can be made to mimic the structure of the extracted function by unrolling tailrec_pre.

$$\begin{aligned}\mathsf{loop\_pre}(x, pos) = \mathsf{if}\ pos = 0\ \mathsf{then}\ \mathsf{loop\_pre}(x{+}1, 1)\ \mathsf{else}\\ \mathsf{if}\ x \,\&\, 1 = 0\ \mathsf{then}\ \mathsf{loop\_pre}(x, 0)\ \mathsf{else}\\ \mathsf{if}\ x{-}2 \neq 0\ \mathsf{then}\ \mathsf{loop\_pre}(x{-}2, 1)\ \mathsf{else}\ \mathsf{true}\end{aligned}$$

When the ARM code requires side conditions, e.g. on alignment of addresses, these appear in the equation for loop_pre. An example of code with such side conditions is explained in Myreen et al. [3].

3.5. Scalability and limitations

As mentioned above, we have implemented this technique for function extraction in HOL4 and applied our implementation to ARM, x86, and PowerPC machine code. Our largest examples are verified Lisp interpreters [6], each approximately 2000 instructions long: this includes code for parsing, printing, and evaluation of Lisp expressions, and copying garbage collection.

This function extractor (decompiler) is reasonably reliable for user-mode code where jumps are only made to relative offsets from the program counter. Procedure call/return are also handled well in most common cases. Our approach is less well suited to code with more sophisticated control flow, e.g. code where jumps are made to code pointers. However, this technique is often still applicable and useful within subcomponents of such code; for example, we found our decompiler very helpful when the verifying a just-in-time compiler [14] (just-in-time compilers make extensive use of code pointers).

The decompilation algorithm presented here is only applicable to programs that have deterministic behaviour. We cannot handle the case when the behaviour of the code is not representable as a function of the starting state.

4. Decompiling with heuristics

The decompiler described above achieves reasonable robustness as a result of not requiring any guesswork to be done as part of the algorithm. However, it also produces very direct translations of the code, and thus produces functions that operate at a very low level of abstraction. In this section, we sketch how decompilation can be made smarter using domain-specific heuristics (at the cost of making the implementation less robust).

4.1. Motivating example: reversal of a linked list

Consider the following C code for destructively reversing a linked list. The loop in this C code successively pops an element off the head of the list pointed to by `i` and makes that element the head of the list stored under pointer `p`. On exit, `p` points to the reversed version of the list originally pointed to by `i`.

```
p = NULL;
while((*i) != NULL) {
  x = (*i)->tl;
  (*i)->tl = p;
  p = (*i);
  (*i) = x;
}
```

Compiling this C code into ARM produces the machine code

```
E3A02000 EA000003 E5910004 E5812004 E1A02001 E1A01000 E3510000 1AFFFFF9
```

By applying the decompiler described above, we get a function mc_reverse which describes the effect of the C code in terms of a memory segment m (the notation $m[r_1 \mapsto r_2]$ denotes function m with the value of register r_1 set to r_2).

$$\mathsf{mc\_reverse}(r_1, m) = \mathsf{let}\ r_2 = 0\ \mathsf{in}\ \mathsf{mc\_rev}(r_1, r_2, m)$$

$$\begin{aligned}\mathsf{mc\_rev}(r_1, r_2, m) = \mathsf{if}\ r_1 = 0\ \mathsf{then}\ (r_1, r_2, m)\ \mathsf{else}\\ \mathsf{let}\ r_0 = m(r_1)\ \mathsf{in}\\ \mathsf{let}\ m = m[r_1 \mapsto r_2]\ \mathsf{in}\\ \mathsf{let}\ r_2 = r_1\ \mathsf{in}\\ \mathsf{let}\ r_1 = r_0\ \mathsf{in}\\ \mathsf{mc\_rev}(r_1, r_2, m)\end{aligned}$$

In some cases, it might be desirable to have a smart decompiler which can produce more abstract functions—i.e. automatically hide the pointer manipulations and produce a function which operates over lists *xs* and *ys* instead of a flat array-like memory *m*.

$$\mathsf{reverse}(xs) = \mathsf{rev}(xs, [])$$

$$\begin{aligned}\mathsf{rev}([], ys) &= ys\\ \mathsf{rev}(x :: xs, ys) &= \mathsf{rev}(xs, x :: ys)\end{aligned}$$

In the next two sections, we show how our decompiler described in Section 3 can be made smart enough to manage this given a simple hint: being told that it should assume that r_1 and r_2 point to linked lists on entry into the loop.

4.2. Special support for linked lists

In order to decompile abstractions *xs* and *ys* from machine code, we must first formally define a shape predicate specifying what it means for a linked list *xs* to be stored in memory. Let *xs* be the list of 32-bit word data stored in the list. We define a recursive list assertion in the style of separation-logic list assertions.

$$\begin{aligned}\mathsf{list}\ a\ [] &= \langle a = 0\rangle\\ \mathsf{list}\ a\ (x :: xs) &= \exists b.\ \mathsf{M}\ a\ b * \mathsf{M}\ (a{+}4)\ x * \mathsf{list}\ b\ xs * \langle a \neq 0\rangle * \langle a\,\&\,3 = 0\rangle\end{aligned}$$

The loop in our example above maintains two linked lists: the address of one list *xs* is kept in register 1 and the address of the other list *ys* is kept in register 2. We can express this formally using the $*$ operator to combine two list assertions and implicitly specify that the two lists do not overlap.

$$\exists a\ b.\ \mathsf{r1}\ a * \mathsf{r2}\ b * \mathsf{list}\ a\ xs * \mathsf{list}\ b\ ys \tag{3}$$

We give this formula as a hint: the decompiler is told that it must express the precondition in terms of (3) at the start of iteration of the loop. This hint allows the decompiler to use basic facts about lists to produce the following two Hoare-triple theorems describing each path through the loop code. One theorem describes the path that exits the loop:

$$\begin{aligned}&(xs = []) \Rightarrow\\ &\{\exists a\ b.\ \mathsf{r1}\ a * \mathsf{r2}\ b * \mathsf{list}\ a\ xs * \mathsf{list}\ b\ ys * \mathsf{r0}\ \_ * \mathsf{pc}\ (p{+}24) * \mathsf{s}\}\\ &p : \mathsf{reverse\_code}\\ &\{\exists b.\ \mathsf{r1}\ 0 * \mathsf{r2}\ b * \mathsf{list}\ b\ ys * \mathsf{r0}\ \_ * \mathsf{pc}\ (p{+}32) * \mathsf{s}\}\end{aligned}$$

A second similar theorem describes one pass through the loop body:

$$\begin{aligned}&(xs \neq []) \Rightarrow\\ &\{\exists a\ b.\ \mathsf{r1}\ a * \mathsf{r2}\ b * \mathsf{list}\ a\ xs * \mathsf{list}\ b\ ys * \mathsf{r0}\ \_ * \mathsf{pc}\ (p{+}24) * \mathsf{s}\}\\ &p : \mathsf{reverse\_code}\\ &\{\exists a\ b.\ \mathsf{r1}\ a * \mathsf{r2}\ b * \mathsf{list}\ a\ (\mathsf{tail}\ xs) * \mathsf{list}\ b\ (\mathsf{head}\ xs :: ys) * \mathsf{r0}\ \_ * \mathsf{pc}\ (p{+}24) * \mathsf{s}\}\end{aligned}$$

These theorems can be used to instantiate the loop rule from Section 3, which produces a function rev:

$$\mathsf{rev}(xs, ys) = \mathsf{if}\ xs = []\ \mathsf{then}\ ys\ \mathsf{else}\ \mathsf{rev}(\mathsf{tail}\ xs, \mathsf{head}\ xs :: ys)$$

List-aware postprocesssing can reformulate this into

$$\begin{aligned}\mathsf{rev}([], ys) &= ys\\ \mathsf{rev}(x :: xs, ys) &= \mathsf{rev}(xs, x :: ys)\end{aligned}$$

By performing similar decompilation of the enclosing code, i.e. the loop initialisation code, we get the top-level function reverse:

$$\mathsf{reverse}(xs) = \mathsf{rev}(xs, [])$$

and a certificate theorem which states that reverse accurately describes the effect of the code. Note that this theorem need not have a side condition on termination of reverse, since termination is in this case an instance of primitive recursion, i.e. a case that can be proved automatically.

$$\begin{aligned}&\{\exists a.\ \mathsf{r1}\ a * \mathsf{r2}\ \_ * \mathsf{list}\ a\ xs * \mathsf{r0}\ \_ * \mathsf{pc}\ p * \mathsf{s}\}\\ &p : \mathsf{reverse\_code}\\ &\{\exists a.\ \mathsf{r1}\ \_ * \mathsf{r2}\ a * \mathsf{list}\ a\ (\mathsf{reverse}(xs)) * \mathsf{r0}\ \_ * \mathsf{pc}\ (p{+}32) * \mathsf{s}\}\end{aligned}$$

4.3. Scalability and limitations

Although our heuristics-enhanced decompilation works well for simple examples such as list reversal, destructive list append, length of a list, etc., this 'smart' decompiler is much less robust than the straightforward decompiler described in Section 3. Our heuristics-enhanced decompilation requires simple hints from the user, or from an external program, and must also have special-purpose support for any data structure it might come across: singly linked lists with one data word, doubly linked lists, xor-linked list, trees, etc. It remains unclear whether heuristics-enhanced decompilation will scale to interesting examples. However, when this style of decompilation is successful, it provides significantly more elegant functions compared with those in Section 3.

5. Applications

Function extraction provides means for inspecting code and for proving formally that the code behaves as intended. However, it can also be used for verifying passes through a compiler. In the rest of this section, we outline how function extraction can be used to implement trustworthy compilation.

5.1. Compilation

In Section 3, we presented a decompiler which extracts functions such as the following from machine code.

$$\mathsf{mod10}(r_1) = \mathsf{if}\ \ r_1 < 10\ \ \mathsf{then}\ \ r_1\ \ \mathsf{else}\ \ \mathsf{let}\, r_1 = r_1 - 10\ \mathsf{in}\ \mathsf{mod10}(r_1)$$

In this section, we show how one can turn this around: we show how functions, such as the mod10 from above, can be compiled into machine code and how one can automatically prove that the generated machine code calculates the original source function, in this case mod10.

The idea is straightforward: for each function f,

1. we generate, with an unverified algorithm, machine code for function f;
2. we use the decompiler to extract a function f′ from the generated code; then,
3. we automatically prove f = f′; thus the certificate theorem, produced in step 2, relates input function f to the generated code.

This style of proof-producing compilation is called translation validation [15]. The code generator, step 1 above, is unverified and may generate untrusted code, but the result of compilation is trustworthy since steps 2 and 3 verify the result, i.e. prove that the generated code is correct.

When steps 1–3 are applied to mod10 from above, a code generator might produce the following ARM code (without the comments /* ... */):

```
E351000A    L:  cmp r1,#10          /* compare r1 with 10          */
2241100A        subcs r1,r1,#10     /* if 10 <= r1, sub 10 from r1 */
2AFFFFFC        bcs L               /*              goto L         */
```

The decompiler extracts the following function g from this ARM code.

$$\mathsf{g}(r_1) = \mathsf{if}\ \ r_1 < 10\ \ \mathsf{then}\ \ r_1\ \ \mathsf{else}\ \ \mathsf{let}\, r_1 = r_1 - 10\ \mathsf{in}\ \mathsf{g}(r_1)$$

In this case, step 3, i.e. proving mod10 = g, is trivial, as mod10 is identical to g.

In general, the code generator does not need to produce code from which the decompiler would extract a function identical to the input function. It suffices to generate some machine code for which step 3 can prove the two functions equivalent. This aspect of the design allows us to incorporate certain lightweight optimisations into step 1: for example, if step 3 normalises names and expands let expressions, then the code generator may change names of registers (i.e. perform register allocation) and can reorder certain instructions (to avoid pipeline stalls).

5.2. Case studies

We have implemented a verifying compiler as described above and used it in a number of case studies, the largest of which are verified Lisp interpreters implemented in ARM, x86, and PowerPC machine code. This section outlines how our Lisp case study [6] made use of the compiler.

Lisp programs operate over a garbage collected heap in which s-expressions live. A Lisp s-expression is, for our Lisp interpreter, either a symbol Sym s where s is a string; a numeric value Num n where n is a natural number; or a pair of s-expressions Dot x y, where x and y are s-expressions. Standard operations, such as car and cons, are defined as follows:

$$\mathsf{car}\ (\mathsf{Dot}\ x\ y) = x \qquad\qquad \mathsf{cons}\ x\ y = \mathsf{Dot}\ x\ y$$

Using a user-defined predicate lisp, which asserts that a heap of s-expressions is present in memory, we can state and prove that certain sequences of machine instructions execute these Lisp primitives. For example, if the s-expressions $v_1 \ldots v_6$ are stored in a heap (expressed as lisp $(v_1, v_2, v_3, v_4, v_5, v_6, l)$) and the s-expression v_1 is a Dot cell, then the machine instruction `ldr r4,[r3]` assigns car v_1 to v_2, such that the postconditions states that there is a valid heap of s-expressions is again present (expressed as lisp $(v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l)$).

$$(\exists x\, y.\ \mathsf{Dot}\ x\ y = v_1)\ \Rightarrow$$
$$\{\mathsf{lisp}\ (v_1, v_2, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ p\}$$
$$p : \texttt{E5934000 [ldr r4,[r3]]}$$
$$\{\mathsf{lisp}\ (v_1, \mathsf{car}\ v_1, v_3, v_4, v_5, v_6, l) * \mathsf{pc}\ (p+4)\}$$

Note that these Hoare triples are very much in the style of those presented in previous sections, except that the above Hoare triple makes the machine code seem to operate over a six-register machine where each register is an s-expression instead of the normal 32-bit word.

By supplying the decompiler with such descriptions for each instruction block supplied to it, the decompiler can produce functions which operate over s-expressions instead of 32-bit words. An even more usable tool is acquired if we let the compiler assemble the code blocks for us. One can supply the compiler with Hoare triples such as the above Hoare triple for car, and it will subsequently be able to compile functions with the line

$$\mathsf{let}\ v_2 = \mathsf{car}\ v_1\ \mathsf{in}$$

directly into ARM machine code. This is the technique which allowed us to build full Lisp implementations in ARM, x86, and PowerPC code: Lisp evaluation (and parsing and printing) were carefully coded up as tail-recursive functions that fit into a subset of operations for which our proof-producing compiler has verified building blocks which it can assembly into verified machine code.

6. Related work

This paper has touched on a number of topics including function extraction, machine-code verification, formalisation of separation logic, and trustworthy (verifying) compilation. We discuss related work separately for each topic.

6.1. Translation of imperative programs into recursive functions

As mentioned in the introduction, the connection between imperative programs and recursive functions has been known almost since the beginning of computer science. McCarthy [1] gave an example which illustrates this link and proved manually that the functional version of the imperative program indeed computes the same value. This idea of loops as functions has appeared frequently in semantics of programs; for example, in denotational semantics, the semantics of loops are defined to be fixed points, which have solutions that are recursive functions. However, for realistic *operational* semantics this connection becomes much less obvious.

There has been some prior work on automatic function extraction for verification and validation by Boyer and Moore ([2], Section 1), Filliâtre [16], Katsumata and Ohori [17], and Pleszkoch et al. [18]. Filliâtre shows how imperative loops can, in type theory, be turned into recursive functions for purposes of verification. Unlike our approach, his requires the code to be annotated with invariants and does not apply the method to low-level languages. Katsumata and Ohori have developed a decompiler, from a small subset of idealised Java bytecode to recursive functions, based on ideas from type theory. The decompiler implementing Katsumata and Ohori methodology has not been verified. It seems that their decompiler would need to be verified or made proof-producing, if its output were to be used in verification. Pleszkoch et al. showed that function extraction can be useful when software is being tested.

The L4.verified project [19] is a notable example of where a connection between C-like code and recursive functions was proved manually: a team spent months proving the correspondence. It might have been possible to automate parts of their proof using techniques described in this paper.

6.2. Machine-code verification

The motivation of function extraction, as explained in this paper, is to aid program verification, in particular verification of low-level programs such as machine-code programs. To see how it aids verification, consider the following example. Given some machine code which we would like to prove calculates modulo 10 of register 1, we first apply the decompiler to the machine code to obtain a function we will call mod10; if we are able to (manually) prove that this function calculates modulo 10 of its 32-bit word input w,

$$\forall w.\ \mathsf{mod10}(w) = w \bmod 10$$

then a simple rewrite of the certificate theorem proved by the decompiler produces a theorem which states that the machine code calculates modulo 10:

$$\{\mathsf{r1}\ r_1 * \mathsf{pc}\ p * \mathsf{s}\}\ \ p : \mathsf{code}\ \ \{\mathsf{r1}\ (r_1 \bmod 10) * \mathsf{pc}\ (p{+}12) * \mathsf{s}\}$$

Other approaches to machine-code verification include symbolic simulation of an operational semantics, applying a programming logic manually, and verification using a verification condition generator (VCG).

Symbolic simulation calculates the result of applying a next-state function to states where registers and memory location have been assigned symbolic values. This method has been successfully applied by the ACL2 community [20–22]; for example, Boyer and Yu used symbolic simulation in pioneering work on verifying Motorola MC68020 machine code obtained by compiling the GNU string library using GCC [23].

Using a programming logic directly on top of the definition of the semantics of the machine code is another approach. Shao's group at Yale [24–26] have used programming logics to verify functional correctness of slightly idealised assembly programs. Foundational proof-carrying code (FPCC) [27] and Typed Assembly Language (TAL) [28,29] also apply special-purpose programming logics, but generally aim to check relatively weak safety properties, while the techniques presented here are concerned with proving full functional correctness.

Verification condition generators (VCGs) have also been used for machine-code verification. VCGs essentially automate large parts of manual Hoare-style proofs: the code, in this case machine or assembly code, is annotated with assertions. The integrity of the VCG is a concern, as practical VCGs tend to be complex [30,31]. Homeier and Martin showed that VCGs can be verified [32] and Matthews et al. has showed that off-the-shelf theorem provers can be used in a way which gives the benefits of a VCG without actually constructing a full VCG [33]. Hardin et al. have applied the technique described by Matthews et al. to machine code for the Rockwell Collins AAMP7G processor [34].

6.3. Formalisation of separation logic

Our certificate theorems are expressed using a version of Hoare triples that incorporates ideas from separation logic [35]. Separation logic is a descendant of Burstall's work on mutable data [36] and emphasises 'local reasoning' as a method to handle the frame problem [37], i.e. how to state and prove that 'nothing else changed'.

There have been number of formalisations of separation logic [38–40]. The formalisation by Weber [38] (in Isabelle/HOL) followed the standard 'fault-avoiding' semantics of separation logic [41], and required a non-trivial proof to establish the frame rule. Our formalisation of machine-code Hoare triples enables the frame rule to be derived by a very simple proof.

6.4. Compilation

The compiler, presented in Section 5, generates code in an unverified manner, but then afterwards automatically checks the correctness of the generated code. Pnueli et al. [15] call this approach *translation validation*, and have proposed a similar method before. Pnueli et al. showed how translation validation can be implemented for a compiler which maps programs in the synchronous multi-clock data-flow language SIGNAL to asynchronous C code. In the spirit of proof-carrying code, Necula [42] showed that translation validation can scale to conventional, moderately optimising compilers such as the GNU C compiler version 2. Pnueli et al. and Necula built their verifiers as standalone tools: Necula's tool consists of over 10,000 lines of trusted Objective CAML code. The compiler described here instead steers an off-the-shelf theorem prover (HOL4) to a proof. Thus our resulting certificate theorems are immediately usable in further automatic and interactive proofs within the framework of a programmable general-purpose theorem prover.

An alternative to producing a proof for each run is to prove the compiler correct once and for all: a formal connection between source and target code is then achieved by simply instantiating the theorem describing the compiler's correctness. A recent, particularly impressive, milestone in compiler verification was achieved by Leroy [13], who proved that his optimising C compiler maps C programs to observationally equivalent PowerPC assembly code.

7. Summary

We have shown how simple imperative programs and machine code can be automatically translated into proof-certified recursive functions. We then showed how such function extraction can aid program verification and how it can be used to implement trustworthy (verifying) compilation.

References

[1] J. McCarthy, Towards a mathematical science of computation, in: IFIP Congress, North-Holland, 1962, pp. 21–28.
[2] R.S. Boyer, J.S. Moore, Program verification, Journal of Automated Reasoning 1 (1) (1985) 17–23.
[3] M.O. Myreen, K. Slind, M.J.C. Gordon, Machine-code verification for multiple architectures—an application of decompilation into logic, in: Formal Methods in Computer-Aided Design, FMCAD, IEEE, 2008.
[4] M.O. Myreen, M.J. Gordon, Transforming programs into recursive functions, in: Brazilian Symposium on Formal Methods, SBMF, Elsevier, 2008.
[5] M.O. Myreen, K. Slind, M.J. Gordon, Extensible proof-producing compilation, in: Compiler Construction, CC, Springer, 2009.
[6] M.O. Myreen, M.J.C. Gordon, Verified LISP implementations on ARM, x86 and PowerPC, in: Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, Springer, 2009, pp. 359–374.
[7] Project sources files available under 'HOL/examples/machine-code' in the HOL4 distribution at SourceForge: http://hol.sourceforge.net/.
[8] K. Slind, M. Norrish, A brief overview of HOL4, in: Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, Springer, 2008.
[9] C.A.R. Hoare, An axiomatic basis for computer programming, Communications of the ACM 12 (10) (1969) 576–580.
[10] P. Manolios, J.S. Moore, Partial functions in ACL2, Journal of Automated Reasoning 31 (2) (2003) 107–127.

[11] A. Fox, Formal specification and verification of ARM6, in: Proceedings of Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, vol. 2758, Springer, 2003, pp. 25–40.
[12] S. Sarkar, P. Sewell, F.Z. Nardelli, S. Owens, T. Ridge, T. Braibant, M.O. Myreen, J. Alglave, The semantics of x86-CC multiprocessor machine code, in: Principles of Programming Languages, POPL, ACM, 2009, pp. 379–391.
[13] X. Leroy, Formal certification of a compiler back-end, or: programming a compiler with a proof assistant, in: Principles of Programming Languages, POPL, ACM, 2006, pp. 42–54.
[14] M.O. Myreen, Verified just-in-time compiler on x86, in: Principles of Programming Languages, POPL, ACM, 2010, pp. 107–118.
[15] A. Pnueli, M. Siegel, E. Singerman, Translation validation, in: Tools and Algorithms for Construction and Analysis of Systems, TACAS, Springer, 1998, pp. 151–166.
[16] J.-C. Filliâtre, Verification of non-functional programs using interpretations in type theory, Journal of Functional Programming 13 (4) (2003) 709–745.
[17] S. Katsumata, A. Ohori, Proof-directed decompilation of low-level code, in: European Symposium on Programming, ESOP, Springer-Verlag, UK, 2001, pp. 352–366.
[18] M.G. Pleszkoch, R.C. Linger, A.R. Hevner, Introducing function extraction into software testing, the data base for advances in information systems: special issue on software systems testing, ACM SIGMIS 39 (3) (2008) 41–50.
[19] S. Winwood, G. Klein, T. Sewell, J. Andronick, D. Cock, M. Norrish, Mind the gap: a verification framework for low-level C, in: Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, Springer, 2009, pp. 500–515.
[20] R.S. Boyer, J.S. Moore, Proving theorems about pure LISP functions, Journal of the ACM 22 (1) (1975) 129–144.
[21] J.S. Moore, Symbolic simulation: an ACL2 approach, in: Formal Methods in Computer-Aided Design, FMCAD, IEEE, 1998, pp. 334–350.
[22] H. Liu, J.S. Moore, Java program verification via a JVM deep embedding in ACL2, in: Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, Springer, 2004, pp. 184–200.
[23] R.S. Boyer, Y. Yu, Automated proofs of object code for a widely used microprocessor, Journal of the ACM 43 (1) (1996) 166–192.
[24] Z. Ni, Z. Shao, Certified assembly programming with embedded code pointers, ACM SIGPLAN Notices 41 (1) (2006) 320–333.
[25] Z. Ni, D. Yu, Z. Shao, Using XCAP to certify realistic system code: Machine context management, in: Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, Springer, 2007, pp. 189–206.
[26] H. Cai, Z. Shao, A. Vaynberg, Certified self-modifying code, in: Programming Language Design and Implementation, PLDI, ACM, 2007, pp. 66–77.
[27] A.W. Appel, Foundational proof-carrying code, in: Logic in Computer Science, LICS, IEEE, 2001.
[28] J.G. Morrisett, D. Walker, K. Crary, N. Glew, From system F to typed assembly language, in: Principles of Programming Languages, POPL, ACM, 1998, pp. 85–97.
[29] J. Chen, D. Wu, A.W. Appel, H. Fang, A provably sound TAL for back-end optimization, in: Programming Language Design and Implementation, PLDI, ACM, 2003, pp. 208–219.
[30] C. Flanagan, J.B. Saxe, Avoiding exponential explosion: generating compact verification conditions, in: Principles of Programming Languages, POPL, ACM, 2001, pp. 193–205.
[31] K.R.M. Leino, Efficient weakest preconditions, Information Processing Letters 93 (6) (2005) 281–288.
[32] P.V. Homeier, D.F. Martin, A mechanically verified verification condition generator, Computer Journal 38 (2) (1995) 131–141.
[33] J. Matthews, J.S. Moore, S. Ray, D. Vroon, Verification condition generation via theorem proving, in: Logic for Programming, Artificial Intelligence, and Reasoning, LPAR, Springer, 2006, pp. 362–376.
[34] D.S. Hardin, E.W. Smith, W.D. Young, A robust machine code proof framework for highly secure applications, in: International Workshop on the ACL2 Theorem Prover and its Applications, 2006.
[35] J. Reynolds, Separation logic: a logic for shared mutable data structures, in: Logic in Computer Science, LICS, IEEE, 2002.
[36] R.M. Burstall, Some techniques for proving correctness of programs which alter data structures, Machine Intelligence 7 (1972) 23–50.
[37] J. McCarthy, P.J. Hayes, Some philosophical problems from the standpoint of artificial intelligence, Machine Intelligence 4 (1969) 463–502.
[38] T. Weber, Towards mechanized program verification with separation logic, in: Computer Science Logic, CSL, in: LNCS, Springer, 2004, pp. 250–264.
[39] H. Tuch, G. Klein, M. Norrish, Types, bytes, and separation logic, in: Principles of Programming Languages, POPL, ACM, 2007, pp. 97–108.
[40] T. Tuerk, A formalisation of Smallfoot in HOL, in: Theorem Proving in Higher Order Logics, TPHOLs, in: LNCS, Springer, 2009, pp. 469–484.
[41] H. Yang, A semantic basis for local reasoning, in: Foundations of Software Science and Computation Structures, FOSSACS, Springer, 2002, pp. 402–416.
[42] G.C. Necula, Translation validation for an optimizing compiler, in: Programming Language Design and Implementation, PLDI, ACM, 2000.